速成教育
SPEED UP EDUCATION

CSCB09

Software Tools and Systems Programming

导师： **VC**

UTSC Week 13 final practice | 2020/4/13

SPEED UP EDUCATION

Example1

Macy's school computer doesn't have `make` installed for some reason! She needs your help writing a shell script that performs a small but important job of `make`. She has a lot of TeX files (filenames end with ".tex"), and there is a program pdflatex for generating corresponding PDF files. Using "A1.tex" as an example, "`pdflatex A1.tex`" generats "A1.pdf". Clearly, she wants to generate if and only if one of:

• A1.tex exists but A1.pdf doesn't

• both exists, but A1.tex is newer

"A1.tex" is just an example—this applies to all TeX and PDF files in the current directory. Implement a Bourne shell script to do this.

The command "`basename A1.tex .tex`" outputs A1 to help you.

Example2

Prof. C and Prof. Y have a lot of PDF files (filenames end with ".pdf"); for some of them, also the compressed versions (filenames end with ".pdf.gz"). Example:

```
$ ls
q2.sh  w.doc  w.doc.gz      x.pdf  x.pdf.gz      y.pdf  z.pdf  z.pdf.gz
```

They have decided that it is safe to delete certain PDF files if the compressed versions exist. They enlist your help to write a Bourne shell script (no bash extra features) for this. However! They want more control, so they will provide PDF pathnames on the command line, and the shell script should consider only those. They may make mistakes, so ignore pathnames that don't exist or don't end with ".pdf" (error messages not required today, not forbidden either). Example:

```
$ sh q2.sh y.pdf x.pdf w.doc
$ ls
q2.sh  w.doc  w.doc.gz  x.pdf.gz  y.pdf  z.pdf  z.pdf.gz
```

Explanation: Consider y.pdf and x.pdf only, ignore w.doc, and don't touch z.pdf; y.pdf stays because there is no y.pdf.gz, x.pdf is gone because there is x.pdf.gz.

If a pathname includes a directory, look for the compressed version there. Example:

```
$ ls
q2.sh         z.pdf           z.pdf.gz
$ ls ../music
tso.pdf       tso.pdf.gz      tsv.pdf
$ sh q2.sh z.pdf ../music/tso.pdf ../music/tsv.pdf
$ ls
q2.sh         z.pdf.gz
$ ls ../music
tso.pdf.gz    tsv.pdf
```

Since there is no ../music/tsv.pdf.gz, don't delete ../music/tsv.pdf.

Example3

If you can look up in RAM quickly, you can look up on disk slowly!

In this question, you will write code for binary search tree lookup, but the tree is stored in a binary file on disk! The file consists of 0 or more nodes defined by this struct:

```
typedef struct {
    int key;
    long left , right;    // file  positions (absolute) or -1L
} node;
```

key is a key as usual. Note that instead of pointers to children, we have file positions (offsets) of children, so their types are long as in `fseek`; accordingly, when there is no left/right child, we use -1L (−1 but type long).

The file may be empty, meaning the tree is empty; but if not, we know that the root node is at the beginning—position 0. Other nodes may be anywhere else, we only know that positions of nodes are non-negative multiples of `sizeof(node).`

Implement lookup:

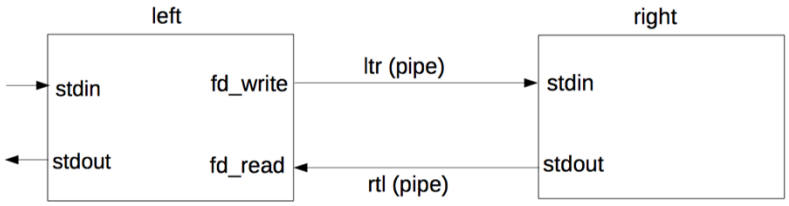
```
int is_present(FILE *f, int needle );
```

This looks for needle in the tree in f. It should return 1 (True) if needle is a key in the tree, 0 (False) if not.

You do not know where the current file position is before this function begins. You may assume that f allows fseek, and fread on f either hits EOF or succeeds. You may assume that if left is non-negative, the left child node exists at that position; similarly for right.

Example4

In this question, you will implement in Q6.c (starter provided) setting up the processes and pipes in this diagram (there is also a parent process lurking, not shown):



The right process will eventually exec a program that isn't aware of this setup except to read from stdin and write to stdout. Therefore before exec, you will set up stdin to be the read end of the ltr pipe, stdout to be the write end of the rtl pipe.

The left process won't exec, instead call

```
void do_left(int fd_read , int fd_write );
```

Therefore you will call `do_left` so that `fd_read` is the read end of the rtl pipe, `fd_write` is the write end of the ltr pipe. Note that the left process inherits stdin and stdout from the parent, and `do_left` can perform I/O on all 4 distinct FDs. Both left and right are child processes of a common parent. The code is structured as follows:

(a) [4 marks] Complete the code for the left child.
(b) [5 marks] Complete the code for the right child.
(c) [2 marks] Complete the code for the parent after forking and before waiting.

```
void leftright(void) {
    // parent begins
    int ltr [2];
    int rtl [2];

    pipe(ltr);
    pipe(rtl);

    pid_t left = fork ();
    if (left == 0) {
        // left child code , part (a)
    } else {
        pid_t right = fork ();
        if (right == 0) {
            // right child code , part (b)
            execlp (" right" , " right" , (char *) NULL);
            exit (1);
        } else {
            // parent code after both forks , part (c)
            wait(NULL);
            wait(NULL);
        }
    }
}
```

Example5

The do_left from the last question

void do_left(int fd_read , int fd_write);

actually does this job:

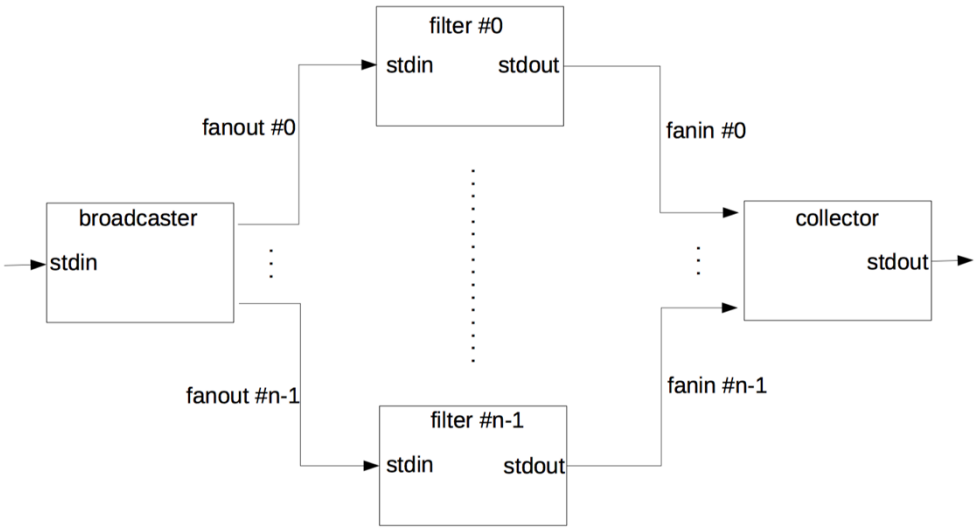
- Copy data from `fd_read` to stdout, verbatim.
- Copy data from stdin to `fd_write` but change all lower case letters to upper case (in the sense of `toupper`).

Since there are two data sources with unknown arrival times, select is a simple way to wait for data, given that the two input FDs won't change.

Implement do_left. Use buffer sizes of 512 bytes. You may assume that write is successful and doesn't block. You may assume fd_read != 0. When any one of the input sources gives EOF, call exit(0).

Example6

In this question, you will implement an ensemble of processes and pipes like in this diagram:



There are n filter processes. Each reads a sequence of real numbers from stdin (which comes from a fanout pipe, but the program doesn't need to know), and outputs a sequence of real numbers to stdout (which goes to a fanin pipe, but the program doesn't need to know). All filter processes run the same program code, but they receive different command line arguments—like biquad from Assignment 3, but just 1 argument instead of 5 for simplicity.

The broadcaster process copies data from its stdin to all n fanout pipes, so that all filter processes receive the same data. But the broadcaster should not care about the data format; it's just a copier.

The collector process sums the sequences from the fanin pipes termwise, and outputs the summed sequence. In other words, it reads one real number from each fanin pipe, adds them, outputs the sum to stdout; repeat.

All of these n + 2 processes have a common parent process (not shown).

The parent process, the broadcaster process, and the collector process run code from the same program—they just run different parts of the program code. This program is structured like:

```
int main(int argc , char ** argv)
{

    // Mostly pseudocode shown.

    variable declarations and setup ---omitted , but see below

    pipe creation , part (a)

    fork
    if child {
        broadcaster code , part (b)
    }

    fork
    if child {
        collector code , part (c)
    }

    fork-exec filters , part (d)

    wait for all children ---omitted
    return 0;
}
```

The following variables have been declared and set as described in the comments.

```
int n;                  // number of filter processes needed
char *filtername;       // pathname of filter program
char *filterarg[n];     // filterarg[i] is the command line argument
                        // for filter #i
```

The following variables have been declared, but you will have to set them:

```
int fanout[n][2], fanin[n][2]; // file descriptors of pipes
```

You may assume that I/O operations and syscalls have no error or failure apart from end of input.

(a) [2 marks] Write C code to create the 2n pipes and store their file descriptors in fanout and fanin.

(b) [5 marks] Write C code for the broadcaster. Use a buffer size of 4096. The process exits when end of input from stdin is hit.

(c) [6 marks] Write C code for the collector. Use double for real numbers. Use the " %g" format for output. Process exit when end of input from any fanin pipe is hit.

(d) [11 marks] Write C code to fork-exec the n filter children, including redirecting stdin and stdout as necessary.

Example7

The following are typical system calls involved when using Internet domain (IPv4) stream sockets, listed in random order:

```
1. c = accept(s, NULL, NULL);
2. connect(s, (struct sockaddr *)&youraddr, sizeof(sockaddr_in));
3. s = socket(AF_INET, SOCK_STREAM, 0);
4. bind(s, (struct sockaddr *)&myaddr, sizeof(sockaddr_in));
5. listen(s, 10);
```

Assume that c and s have been declared, and myaddr and youraddr have been declared and filled in properly.

(a) [5 marks] Give the correct order for a server, and indicate which system call should be repeatedly made in a loop if the server anticipates multiple clients. You may write just the numbers or the syscall names.

(b) [2 marks] Give the correct order for a client (that just needs one connection to one server). You may write just the numbers or the syscall names.

(c) [13 marks] A modern web browser (as a client) makes multiple concurrent connections to servers and downloads from them concurrently. Implement a simple version of this feature.

Two sockets s1 and s2 are already created and connected to two servers, and web page requests have already been sent; you just have to read from them now. But use select for multiplexing so you can work on any data from either side as soon as it arrives. You may assume s1 < s2. Use a buffer size of 1600 for reading. Since either side may finish much earlier than the other, each socket must be closed as soon as end of input is reached, and should not be given to select again.

Two functions are provided to process the data received:

void worker1(const char *buf, size_t n); void worker2(const char *buf, size_t n); Call worker1 for every chunk of data from s1; likewise for worker2 and s2.